CRT-2001-22

Centre de recherche sur les transports (C.R.T.)
Centre for Research on Transportation

THE STRETCH CONSTRAINT

by

Gilles Pesant

May 2001

102-02942/

PUBLICATION
CRT-2001-22

THE STRETCH CONSTRAINT*

by

Gilles Pesant[1]

*This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche (FCAR).

[1] École Polytechnique de Montréal and Centre de recherche sur les transports, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal, Canada H3C 3J7

Dépôt légal – Bibliothèque nationale du Québec
Bibliothèque nationale du Canada, 2001

Résumé. Cet article décrit un algorithme de filtrage pour un type de contrainte fréquemment présente dans les problèmes d'emploi du temps mais qui possède également un champ d'application plus vaste. Cet algorithme est adapté au paradigme de la programmation par contraintes. Définie sur une suite de variables, la contrainte "stretch" restreint le nombre de valeurs identiques consécutives dans la suite. L'algorithme procède en déterminant des intervalles à l'intérieur desquels une sous-suite de valeurs identiques donnée doit se retrouver puis en raisonnant à propos de ceux-ci afin d´éliminer des valeurs du domaine de certaines variables. L'algorithme a une faible complexité en temps de calcul alors que des expérimentations indiquent qu'il est très utile pour guider la recherche d'une solution.

Mots clés. programmation par contraintes, algorithme de filtrage, problèmes d'emploi du temps, horaires cycliques.

Abstract. This paper describes a filtering algorithm for a type of constraint that often arises in rostering problems but that also has wider application. This algorithm is meant to be used within a constraint programming framework. Defined on a sequence of variables, the stretch constraint restricts the number of consecutive identical values in the sequence. The algorithm mainly proceeds by determining intervals in which a given stretch must lie and then reasoning about them to filter out values from the domain of variables. It is shown to have low time complexity and significant pruning capability as evidenced by experimental results.

Key words. constraint programming, filtering algorithm, rostering problems, rotating schedules.

Introduction

This paper describes a filtering algorithm for a type of constraint that often arises in rostering problems. This algorithm is meant to be used within a constraint programming framework [6].

We recall very briefly some main characteristics of constraint programming. The application of constraint programming to the resolution of combinatorial problems is based on modeling those problems using finite domain variables. For each such variable, the (finite) set of all possible values it may take, called its domain, is saved and updated as the computation progresses. Constraints are expressed about the combinations of values these variables may take and are enforced using specialized filtering algorithms, either built-in or user defined, that work by removing inconsistent values from the domains of variables. Ideally, the algorithm should remove all inconsistent values, that is, those for which no combination of values for the rest of the variables can be found that satisfies the constraint, but this cannot always be achieved with reasonable efficiency. The constraints (or, more precisely, their algorithms) interact through shared variables (or, more precisely, their domains). Accordingly, the stretch constraint is implemented through an algorithm removing inconsistent values from domains.

The rest of the paper is organized as follows. The next section briefly describes the usual context in which the constraint is found. Section 2 presents a formulation of the constraint while section 3 explains the filtering algorithm which is used to enforce it. Some experimental results are then reported in section 4 to assess the algorithm's efficiency. Finally, section 5 presents concluding remarks on the applicability of such a constraint.

1 Rostering

Many industries and public services operate around the clock, seven days a week. In such a context, every day a number of work shifts must be covered by one or a team of workers. A workload requirement matrix is usually given or computed which specifies the number of (teams of) workers required for each type of shift of every day, either as a precise value or an interval. For shift work, it is often preferable to schedule work stretches of the same type of shift for individuals: it is easier on their internal body clock as well as on their family and social life. [3].

Accordingly, restrictions on the length of a work stretch are usually given as part of the formulation of the problem. Work stretches will typically be constrained to span at least one or two shifts and at most six or seven. Such a restriction will rarely vary between shift types except maybe for days off. Legal patterns of shift types are also usually given. As a special case of this, a common restriction is that between two consecutive work stretches, some minimum number of days off should be given. Other constraints are present too that do not concern us here.

Week	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	-	-	-	D	D	D	D
2	-	-	E	E	E	-	-
3	D	D	D	-	-	E	E
4	E	E	-	-	N	N	N
5	N	N	N	N	-	-	-

Table 1: A simple rotating schedule. Symbols "D", "E", and "N" indicate day, evening, and night shifts respectively whereas "-" indicates a day off.

1.1 Rotating Schedules

When the personnel is interchangeable, rotating schedules, a repeating pattern of sequences of work and rest days alternating over several weeks, are particularly well adapted. A schedule is given over a cycle of w weeks and the workforce is divided into w teams: initially the first team follows the schedule of week 1, the second one the schedule of week 2, and so forth. After the seventh day, each moves to the next week, the team on week w moving up to week 1. In effect, everyone has an identical schedule but that is out of phase with the other teams. This ensures that everybody is treated equally. An example of a rotating schedule is given in table 1.

That example features stretches of length four and three for day shifts; three and four for evening shifts; seven for night shifts; two, two, two, two, and six for days off.

1.2 Personalized Schedules

When members of the personnel have individual restrictions or preferences that must be taken into consideration, such as unavailabilities due to other activities, rotating schedules become inappropriate. Personalized schedules for each member of personnel are then elaborated. This is typical of some category of personnel such as physicians.

Instead of a cyclic schedule as before, many individual rosters need to be designed, spanning a given scheduling horizon. Constraints on the length of work stretches are still relevant in this context and may vary from one individual to another.

2 The Stretch Constraint

Let work shifts be numbered consecutively from 0 to $n-1$.[1] Consider a constraint programming model for the rostering problem in which a sequence of decision variables $s_0, s_1, \ldots, s_{n-1}$ stand for consecutive work shifts either representing the whole roster, in the case of rotating schedules, or one individual

[1] We choose to start with 0 in order to simplify subsequent modulo expressions.

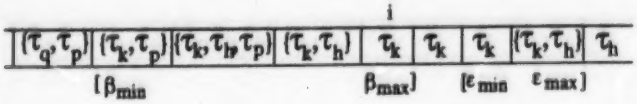

Figure 1: Bounding a stretch of τ_k's on a sample fragment of a schedule.

roster, in the case of personalized schedules. These variables will take their value from the set of shift types $\{\tau_1, \tau_2, \ldots, \tau_m\}$, including one corresponding to a day off.

Definition: Subsequence $s_i, s_{i+1}, \ldots, s_j$ is called a *stretch* when $s_i = s_{i+1} = \cdots = s_j$ but $s_{i-1} \neq s_i$ and $s_{j+1} \neq s_j$.

Let $\check{\lambda}$ and $\hat{\lambda}$ be integer vectors of length m and γ be a boolean value. The stretch constraint may then be formulated as

$$\texttt{stretch}(\langle s_0, s_1, \ldots, s_{n-1}\rangle, \check{\lambda}, \hat{\lambda}, \gamma)$$

with the following semantics:

$\forall\, 0 \leq i \leq n-1$, if $s_i = \tau_k$ then the length of the stretch through it $\in [\check{\lambda}_k, \hat{\lambda}_k]$

The value *true* for γ indicates a cyclic schedule where s_{n-1}'s successor in the sequence is s_0; the value *false* indicates a sequence with no wrap around.

A similar constraint is the sequencing constraint [7]. Defined on a sequence of variables as well, it is used to specify minimum and maximum numbers of appearances of each value within every subsequence of a given length. The main difference is that these values do not have to appear consecutively (i.e. in a stretch).

3 Its Filtering Algorithm

This section is devoted to the detailed description of the algorithm enforcing the semantics of `stretch`. In describing the algorithm, we will take the point of view of rotating schedules for which the sequence of shifts is cyclic. The span of a stretch from indices i to j, denoted $span(i, j)$, is defined as $1 + (j - i) \bmod n$. Also, let $\mathcal{D}_{s_i}$ denote the domain of s_i.

3.1 Determining bounds on a stretch

All the filterings described here are based on information about the possible beginning and end of a particular stretch. For any given shift s_i taking value τ_k, we wish to compute the tightest intervals $[\beta_{min}, \beta_{max}]$ and $[\epsilon_{min}, \epsilon_{max}]$ in which the beginning and the end of the stretch through that shift must lie, respectively,

given the current domains of the s_j's. Figure 1 provides an example. The extremal values of the intervals are derived using the algorithms given below.

$\beta_{\max}$:

1. $j \leftarrow (i-1) \bmod n$;
2. **while** $s_j = \tau_k$ **do**
   a. $j \leftarrow (j-1) \bmod n$;
3. $\beta_{\max} \leftarrow (j+1) \bmod n$;

This first algorithm simply scans the shift variables backwards from index i until it reaches one that is not currently instantiated to τ_k. $\beta_{\max}$ is then set to the index following that (see figure 2). We do not reproduce the algorithm for $\epsilon_{\min}$ here as it is simply the mirror image of the previous one.

$\beta_{\min}$:

1. **if** $\hat{\lambda}_k \geq n$ **then** too_far $\leftarrow (\epsilon_{\min}+1) \bmod n$;
2. **else** too_far $\leftarrow (\epsilon_{\min} - \hat{\lambda}_k) \bmod n$;
3. $j \leftarrow (\beta_{\max}-1) \bmod n$;
4. done $\leftarrow$ false;
5. **while** $j \neq$ too_far **and not** done **do**
   a. **while** $\tau_k \in \mathcal{D}_{s_j}$ **and** $|\mathcal{D}_{s_j}| > 1$ **and** $j \neq$too_far **do**
      i. $j \leftarrow (j-1) \bmod n$;
   b. **if** $\tau_k \notin \mathcal{D}_{s_j}$ **then**
      i. $\beta_{\min} \leftarrow$ find-frontier$(j, k, \beta_{\max})$;
      ii. done $\leftarrow$ true;
   c. **else if** $|\mathcal{D}_{s_j}| = 1$ **then**
      i. $j' \leftarrow j$;
      ii. **while** $s_j = \tau_k$ **and** $j \neq$ too_far **do**
         $j \leftarrow (j-1) \bmod n$;
      iii. **if** $s_j = \tau_k$ **then**
         $\beta_{\min} \leftarrow$ find-frontier$((j'+1) \bmod n, k, \beta_{\max})$;
         done $\leftarrow$ true;
6. **if not** done **then** $\beta_{\min} \leftarrow$ find-frontier$(j, k, \beta_{\max})$;

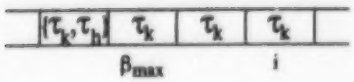

Figure 2: Determining β_{max} on a sample fragment of a schedule.

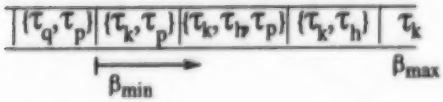

Figure 3: Step 5b in determining β_{min} on a sample fragment of a schedule.

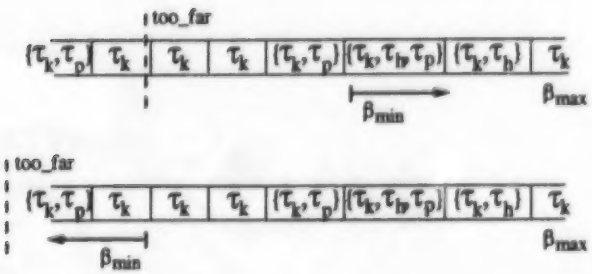

Figure 4: Step 5c in determining β_{min} on a sample fragment of a schedule.

The computation of $\beta_{\min}$ is more involved. In order to avoid unnecessary work a threshold value, *too_far*, is determined beyond which $\beta_{\min}$ cannot possibly lie. Step 1 deals with a special case whereas step 2 corresponds to the general case: the threshold is equal to the earliest position at which the stretch may end minus the longest it may run. In step 3, j is set to the position immediately preceding $\beta_{\max}$. Step 5 iterates until either $\beta_{\min}$ is determined or the threshold is reached. First, the shift variables are scanned backwards from j as long as the current variable has several values in its domain, including τ_k, and the threshold has not been reached (5a). If the scan was stopped because τ_k does not belong to the domain (5b), then $\beta_{\min}$ cannot lie beyond this point but its proper value is determined by the call to *find-frontier*, which we shall describe shortly, and then we are done (see figure 3). Failing that, if the scan was stopped because the shift variable is bound (5c), then it must be that $s_j = \tau_k$. If position j is included in the stretch then so must its immediate predecessor if it is also bound to τ_k, and so on, since a stretch clearly cannot be flanked by shifts of the very same type, from its definition. So, the shift variables are scanned backwards until one that is not currently instantiated to τ_k is met or the threshold is reached (5cii). If at the threshold while still being instantiated to τ_k (5ciii), the stretch would be too long and so $\beta_{\min}$ must lie to the right of that run of bound variables, as determined by the call to *find-frontier*, and we are done. Otherwise a new iteration of step 5 is begun (see figure 4). Finally, if the threshold is reached without encountering any limitation to $\beta_{\min}$, *find-frontier* is called in order to determine $\beta_{\min}$ more precisely in light of what lies just beyond that threshold.

Just as for $\epsilon_{\min}$, the computation of $\epsilon_{\max}$ is not described since it proceeds similarly.

find-frontier(j, k, p):

1. while $j \neq p$

  a. for each $\tau_\ell \in \mathcal{D}_{s_j}$ **such that "**$\tau_\ell\tau_k$**" is a legal pattern**

    i. found $\leftarrow$ true;

    ii. for $i = 1$ **to** $\bar{\lambda}_\ell - 1$

      if $\tau_\ell \notin \mathcal{D}_{s_{(j-i) \bmod n}}$ **then** found $\leftarrow$ false;

    iii. if found **then return** $(j + 1) \bmod n$;

  b. $j \leftarrow (j + 1) \bmod n$;

2. return -1;

find-frontier ensures that on the immediate left of $\beta_{\min}$, enough variables have a common value in their domains to allow a valid neighbouring stretch. Step 1 scans the variables forward from j until a value τ_ℓ is found that lies in the domains of $s_j, s_{j-1}, \ldots, s_{j-\bar{\lambda}_\ell}$ and that forms a legal pattern of shift types with τ_k, at which time value $(j+1)$ mod n is returned. As an example, consider

the top fragment in figure 4 with $\check{\lambda}_p = 2$: $\beta_{\max} - 1$ will be returned. In step 2, if p is reached without any such value being found then -1 is returned. In the context of the computation of $\beta_{\min}$, the latter means that interval $[\beta_{\min}, \beta_{\max}]$ is empty, triggering one of the filtering rules in section 3.3.

The following confirms that our later reasoning based on those intervals will be sound:

Theorem: Given the current domains of the sequence of variables, intervals $[\beta_{\min}, \beta_{\max}]$ and $[\epsilon_{\min}, \epsilon_{\max}]$ as computed above must respectively include the starting and the ending indices of the stretch through s_i.

Clearly, the starting index cannot be larger than $\beta_{\max}$ since by construction of the latter this would mean that the stretch has a shift of the same type as its immediate left neighbour, contradicting its definition. The argument that the starting index cannot be smaller than $\beta_{\min}$ has already been given while describing the corresponding algorithm. Similar arguments can be made for $\epsilon_{\min}$ and $\epsilon_{\max}$, thus completing the proof. □

Unfortunately, these intervals are not necessarily the tightest as it is possible to construct examples that would require examining the whole sequence instead of just a fragment of it as we do. However that alternative would be more computationally expensive, as we discuss below.

It is easy to see that all the algorithms given above, with the exception of the work performed in *find_frontier*, exhibit a worst-case time complexity that is linear in $\hat{\lambda}_k$, the maximum length of a stretch of type τ_k. Because of the three nested loops in *find_frontier*, its worst-case time complexity is in $\mathcal{O}(\hat{\lambda}_k \cdot m \cdot \max\{\check{\lambda}_i : 1 \leq i \leq m\})$. This is still low since in practice the number of shift types seldom exceeds four and the maximum length of stretches rarely exceeds eight.

Before proceeding further, we give a brief overview of the filtering algorithm. Two types of events are considered: when a value is removed from the domain of a shift variable, potentially breaking a stretch of the corresponding shift type, the possibility of a valid stretch of that type is verified on both sides of the variable; when a variable becomes bound, several filterings are applied based on where the beginning and end of the stretch through that variable may lie.

3.2 Detecting broken stretches

Each time the domain of a shift variable s_i is modified, the following algorithm may be applied:

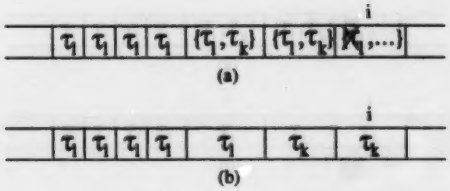

Figure 5: Detecting a broken stretch (a); a possible fragment of the schedule (b).

F1.

for each τ_ℓ just removed from $\mathcal{D}_{s_i}$

1. **if $\tau_\ell \in \mathcal{D}_{s_{(i-1) \bmod n}}$**
   a. $j \leftarrow$ computation of $\beta_{\min}$ for a potential stretch of τ_ℓ's ending at $(i-1) \bmod n$;
   b. **if** $\text{span}(j, (i-1) \bmod n) < \check{\lambda}_\ell$ **or** $j = -1$ **then**
      i. **remove τ_ℓ from $\mathcal{D}_{s_{(i-1) \bmod n}}$;**
2. **if $\tau_\ell \in \mathcal{D}_{s_{(i+1) \bmod n}}$**
   a. $j \leftarrow$ computation of $\epsilon_{\max}$ for a potential stretch of τ_ℓ's starting at $(i+1) \bmod n$;
   b. **if** $\text{span}((i+1) \bmod n, j) < \check{\lambda}_\ell$ **or** $j = -1$ **then**
      i. **remove τ_ℓ from $\mathcal{D}_{s_{(i+1) \bmod n}}$;**

The algorithm considers in turn each value τ_ℓ removed from the domain of s_i. If a stretch of τ_ℓ's may appear on the immediate left of s_i, the left-hand side is examined (step 1). Step 1a determines the earliest beginning of such a stretch. If it cannot be long enough to be valid then value τ_ℓ is removed from the domain of $s_{(i-1) \bmod n}$. Note that, at this time, we may not remove that value for further neighbours to the left since it might prevent a stretch ending a little before. For example, consider figure 5a with $\check{\lambda}_\ell = 3$ and $\hat{\lambda}_\ell = 5$: value τ_ℓ will be removed from $\mathcal{D}_{s_{(i-1) \bmod n}}$ but τ_ℓ is still possible for $s_{(i-2) \bmod n}$, as shown in figure 5b. Nevertheless, that single removal in turn may trigger further deletions. The examination of the right-hand side (step 2) proceeds similarly.

The application of this algorithm guarantees that any value left in the domain of a shift variable has enough peers in neighbouring shift variables to make up a minimum length stretch. This property simplifies the algorithm for filtering rules F5 and F6 in section 3.3.

Since F1 features in the worst case $\mathcal{O}(m)$ computations of $\beta_{\min}$ which itself

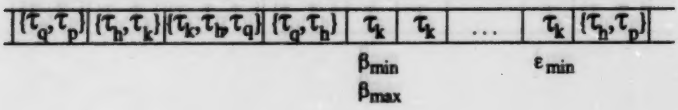

Figure 6: Illustration of rule F5.

includes a call to *find-frontier*, its worst-case time complexity is in $\mathcal{O}(\hat{\lambda}_k \cdot m^2 \cdot \max\{\check{\lambda}_i : 1 \leq i \leq m\})$.

3.3 Reasoning on the potential extent of a stretch

Once we have the intervals, a number of filtering rules may be applied each time a shift variable is instantiated to a value τ_k. First, a value of -1 may have been returned by a call to *find-frontier*:

F2. If $\beta_{\min} = -1$ or $\epsilon_{\max} = -1$ then the stretch constraint is violated.

We may also simply discover that the stretch is necessarily too long or too short:

F3. If $\text{span}(\beta_{\max}, \epsilon_{\min}) > \hat{\lambda}_k$ then the stretch constraint is violated.

F4. If $\text{span}(\beta_{\min}, \epsilon_{\max}) < \check{\lambda}_k$ then the stretch constraint is violated.

F5. If $\beta_{\max} = \beta_{\min}$ then we know precisely where the stretch begins. Value τ_k may thus be removed from preceding shift variables:

1. $\lambda \leftarrow \infty$;
2. **for each** $\tau_\ell \in \mathcal{D}_{s_{(\beta_{\max}-1) \bmod n}}$ **such that** "$\tau_\ell\tau_k$" **is a legal pattern**
   a. **if** $\check{\lambda}_\ell < \lambda$ **then** $\lambda \leftarrow \check{\lambda}_\ell$;
3. **if** $\lambda \neq \infty$ **then for** $i = 1$ **to** λ
   a. **remove** τ_k **from** $\mathcal{D}_{s_{(\beta_{\max}-i) \bmod n}}$;

Step 2 computes the length λ of the shortest feasible neighbouring stretch on the left so that step 3 may remove value τ_k from the λ shift variables immediately preceeding the current stretch since those neighbours must necessarily be of a different type. For example in figure 6, suppose $\check{\lambda}_h = 3$ and $\check{\lambda}_q = 2$. So $\lambda = 2$ and τ_k may be removed from the domain of $s_{\beta_{\max}-2}$.

F6. If $\epsilon_{\max} = \epsilon_{\min}$ then we know precisely where the stretch ends. Value τ_k may thus be removed from following shift variables, using an algorithm that is the symmetric counterpart of the previous one.

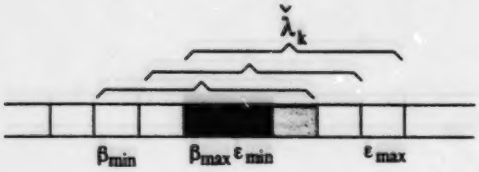

Figure 7: The slots in dark gray are already fixed. The one in light gray can also be fixed, from rule F7.

F7. Given the minimum length of the stretch, it may sometimes be the case that wherever it lies within the interval $[\beta_{\min}, \epsilon_{\max}]$, a particular position is always covered by that stretch:

> **1. for** $i = 1$ **to** $\check{\lambda}_k - \text{span}(\beta_{\min}, \epsilon_{\min})$
>
> a. $s_{(\epsilon_{\min}+i) \bmod n} \leftarrow \tau_k$;
>
> **2. for** $i = 1$ **to** $\check{\lambda}_k - \text{span}(\beta_{\max}, \epsilon_{\max})$
>
> a. $s_{(\beta_{\max}-i) \bmod n} \leftarrow \tau_k$;

Step 1 fixes the variables beyond $\epsilon_{\min}$ which would be included in the shortest possible stretch (of length $\check{\lambda}_k$) starting at the earliest possible position ($\beta_{\min}$), if any, because these variables must be part of any valid stretch (see figure 7). Step 2 does the same from the other end.

The worst-case time complexity of the first three rules is in $\Theta(1)$, that of rules F5 and F6 is in $\mathcal{O}(m+\max\{\check{\lambda}_i : 1 \leq i \leq m\})$, and in $\mathcal{O}(\max\{\check{\lambda}_i : 1 \leq i \leq m\})$ for F7. This concludes our complexity analysis — note that the overall complexity of the algorithm is not related to n.

4 Experimental Results

In this section we propose to evaluate the efficiency of our algorithm both on realistic problems and on a larger set of generated benchmarks.

The stretch constraint as described in this paper was used to model and solve several real-life rotating schedule problem instances from the literature ([3][1][5][2]) using an algorithm described in [4]. Though it is far from being the only reason for the success of the algorithm since a few other global constraints are present and a specially tailored search strategy was devised, its impact on the overall efficiency is significative, as shown in table 2. Seven versions of the filtering algorithm were tested: a complete one, five versions leaving out

version of stretch	alcan	borot	MOT	butler	laporte	hung	lau
complete	1004	16	0	2024	1	100	1
without F1	1491	67	0	11422	19	88	1
without F2	1004	16	0	2024	1	88	1
without F4	1004	16	0	2024	1	100	1
without F5,F6	1180	50	1	4831	1	100	48
without F7	1529	55	161	184685	121	100	1
naive	3578	178	204	-	457587	88	59

Table 2: Number of failed branches for different versions of the algorithm applied to cyclic rostering problems from the literature.

length	# values	mean gap	complete		naive	
			fails	time	fails	time
25	3	1	10.8	0.01	4970.6	0.74
		3	1.8	0.01	206.2	0.05
		5	1.9	0.02	401.8	0.04
	5	1	138.8	0.13	948310.0	212.52
		3	4.8	0.02	49323.9	3.53
		5	4.8	0.02	76988.1	8.92
	7	1	13.3	0.03	386865.0	73.14
		3	55.7	0.05	1.36×10^6	240.19
		5	23.0	0.03	179134.0	32.34
50	3	1	31.2	0.04	-	-
		3	5.7	0.04	-	-
		5	1.9	0.04	586220.0	105.36
	5	1	55.4	0.04	-	-
		3	158.6	0.11	-	-
		5	74.6	0.03	-	-
	7	1	914.7	0.38	-	-
		3	2532.2	0.77	-	-
		5	37114.3	1.97	-	-

Table 3: A comparison of the number of failed branches and computation time (in seconds on a SUN Ultra 5 at 400MHz) between the complete and naive versions over a range of generated benchmarks.

one particular filtering component[2], and a naive version. Each time a shift variable is bound, the latter passively checks that the current stretch through that variable is not too long and that there is sufficient room to reach minimum length. For each version of the `stretch` constraint and each problem instance, the number of failures is reported.

A first observation is that every instance is easily solved by the complete version, but not so easily by the naive version. In fact, one instance (butler) could not be solved within one hour of computation. Of the individual filtering rules tested, F1 and F7 appear to be the most crucial while F2 and F4 have little effect. In fact, F4 is redundant with respect to F1 or F7.

For a more thorough assessment of the `stretch` constraint, a set of instances of the following benchmark problem were generated: given the length of a circular sequence, the set of values that may be used in the sequence, as well as minimum and maximum lengths for a stretch of each of those values, find a valid sequence. Here every juxtaposition of shift types constitutes a legal pattern. Granted, this is not a difficult problem if we proceed sequentially and with a bit of planning but we further impose that the sequence should be filled in random order with values randomly selected from the current domain (the pseudo-random number generator used the same seed for both versions of the

[2]F3 was not left out because it is necessary for the correctness of the algorithm.

constraint). This not only makes the problem harder to solve but also approximates a more realistic context in which fragments of the sequence may be preassigned or fixed through the intervention of other constraints.

The first three columns of table 3 describe the parameters of the instances. The mean gap (column 3) refers to the difference between the minimum and maximum allowed stretch-lengths for a given value. In the following four columns, each entry corresponds to an average over ten instances. Entries left blank indicate that the corresponding instances could not be solved within one hour of computation.

It is difficult to notice a clear trend across the parameter space. With the exception of the occasional harder instance, it appears that the difficulty increases with the number of values allowed and, obviously, with the length of the sequence. The effect of the mean gap is unclear. More interestingly, the full version of the `stretch` constraint performs up to several orders of magnitude better than the naive version both in the size of the search tree and the computation time, even though more effort is expended during each call to the former. The smaller variance in the results of the full version is also noteworthy.

5 Conclusion

This paper presented a new global constraint on a sequence of variables. It may be useful whenever limits are given on the number of consecutive identical values in the sequence. One immediate domain of application is rostering and several supporting experiments in that area were reported. The filtering algorithm used by the constraint was shown to have low complexity and significant pruning capability.

This constraint was successfully used in the multi-shift scheduling system described in [4] to model several of the constraints sometimes found in rostering problems: constraints on the length of work stretches of a given type or of mixed types, constraints on the length of stretches of days off, constraints on the number of consecutive weekends off, etc. It was also instrumental in constraining the number and spacing of stretches of each length, through a simple extension.

Acknowledgements

The author would like to thank Gilbert Laporte for introducing him to cyclic rostering.

References

[1] N. Balakrishnan and R.T. Wong. A Network Model for the Rotating Workforce Scheduling Problem. *Networks*, 20:25–42, 1990.

[2] R. Hung. A Multiple-Shift Workforce Scheduling Model under the 4-Day Workweek with Weekday and Weekend Labour Demands. *Journal of the Operational Research Society*, 45:1088–1092, 1994.

[3] G. Laporte. The Art and Science of Designing Rotating Schedules. *Journal of the Operational Research Society*, 50:1011–1017, 1999.

[4] G. Laporte and G. Pesant. A General Multi-Shift Scheduling System. Working paper, 2001.

[5] H.C. Lau. Combinatorial Approaches for Hard Problems in Manpower Scheduling. *Journal of the Operations Research Society of Japan*, 39:88–98, 1996.

[6] K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[7] J.-C. Régin and J.-F. Puget. A Filtering Algorithm for Global Sequencing Constraints. In *Principles and Practice of Constraint Programming - CP97: Proceedings of the Third International Conference*, pages 32–46. Springer-Verlag LNCS 1330, 1997.